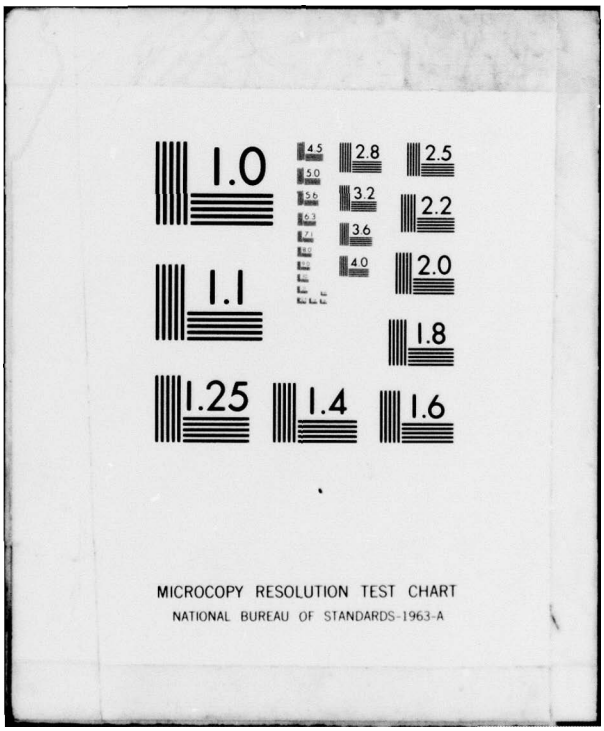
AD-A077 365 NAVAL RESEARCH LAB WASHINGTON DC F/G 12/1
GUIDE TO VECTORIZATION ON THE NAVAL RESEARCH LABORATORY'S TEXAS--ETC(U)
NOV 79 B BROOKS , H BROCK , M MILLER
UNCLASSIFIED NRL-MR-4102-VOL-1 NL

1 OF 1
AD A077365

END
DATE FILMED
1-80
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

LEVEL II

NRL Memorandum Report 4102

Guide to Vectorization on the Naval Research Laboratory's Texas Instruments Advanced Scientific Computer

Volume 1 Vectorization Primer

BARBARA BROOKS AND HARVEY BROCK

Software Systems and Support Branch
Research Computation Center

AND

MAX MILLER

University of Texas at Austin
Austin, Texas 78712

AD A077365

DDC
RECEIVED
NOV 28 1979
A

DDC FILE COPY

November 13, 1979

NAVAL RESEARCH LABORATORY
Washington, D.C.

Approved for public release; distribution unlimited.

79 11 28 041

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NRL Memorandum Report 4102	2. GOVT ACCESSION NO. (14) NRL-MR-4102-VOL-1
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): GUIDE TO VECTORIZATION ON THE NAVAL RESEARCH LABORATORY'S TEXAS INSTRUMENTS ADVANCED SCIENTIFIC COMPUTER. VOLUME 1. VECTORIZATION PRIMER	5. TYPE OF REPORT & PERIOD COVERED: Interim report on a continuing NRL problem.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Barbara Brooks, Harvey Brock and Max Miller*	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Research Laboratory, Washington, DC 20375	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: NRL General and Admin. Function 177215
11. CONTROLLING OFFICE NAME AND ADDRESS (12) 25 (11) 13	12. REPORT DATE: November 19, 1979
	13. NUMBER OF PAGES: 25
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

*Affiliated with: University of Texas at Austin, Applied Research Laboratories, P. O. Box 8029 Austin, Texas 78712

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Vectorization
Advanced Scientific Computer
NX compiler
Vector computer
Parallel processor

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Vectorization Primer is for programmers newly using the Texas Instruments Advanced Scientific Computer (ASC) and it is for managers, scientists, and programmers who are contemplating the ASC.

The concept of vectorization is introduced and illuminated. Fundamental principles and basic programming techniques for achieving vectorization are presented. Problems and pitfalls
—> next page
(Continues)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

251 950

20. Abstract (Continued)

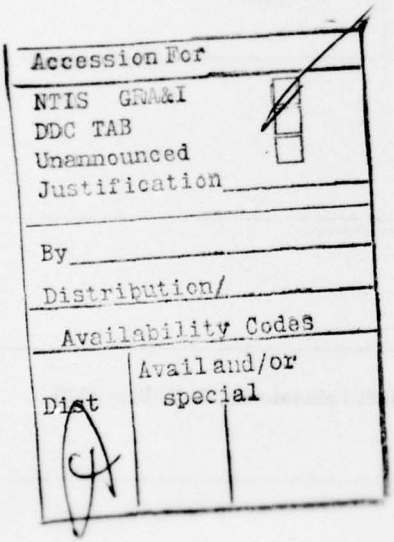
cont.

that FORTRAN programmers, new to the ASC, encounter in writing and vectorizing FORTRAN programs are discussed.

This information is useful to the potential ASC user for weighing the benefits of vectorization against the costs of vectorization in programming effort.

This information assists the new ASC programmer in mastering concepts and techniques for vectorization. It guides him/her around irksome pitfalls.

Volume II of the Guide to Vectorization on the Naval Research Laboratory's Texas Instruments Advanced Scientific Computer is for experienced ASC programmers. Advanced vectorization techniques, examples, and considerations are presented.



CONTENTS

INTRODUCTION 1

1. VECTORIZATION 2

2. PROGRAMMING FOR VECTORIZATION 4
   - 2.1 Conditionality 4
   - 2.2 Subroutine Organization for Array Processing 7
   - 2.3 Vector Math Library Functions 8
   - 2.4 Algorithms and Mathematical Methods 8
   - 2.5 Memory Management 9
   - 2.6 Vector Temporary Space 10
   - 2.7 Using Two Pipes (Parallel Processing) 12
   - 2.8 The Vector Dot Product Instruction 13
   - 2.9 Summary of Programming Principles for Vectorization on the ASC 14

3. MEET THE VECTORIZING COMPILER 15
   - 3.1 Brief Description of ASC CP Architecture 15
   - 3.2 Brief Description of the NX Compiler 16
   - 3.3 Pitfalls for The Unaware Programmer 19
   - 3.3.1 The Timing Routine Pitfall 19
   - 3.3.2 The Local-Variable-In-Register Pitfall 19
   - 3.3.3 The 65, 535 and 32,767 Limitation 20
   - 3.3.4 The 511 Vector Parameter File Limitation 20
   - 3.3.5 Compiler Dumps 20
   - 3.3.6 Compiler Bugs 21
   - 3.4 Conclusion 21

BIBLIOGRAPHY AND REFERENCES 22

ACKNOWLEDGMENT 22

GUIDE TO VECTORIZATION ON

THE NAVAL RESEARCH LABORATORY'S

TEXAS INSTRUMENTS ADVANCED SCIENTIFIC COMPUTER

Introduction

The vector computer at the Naval Research Laboratory is capable of performing up to 48 million floating point operations per second. Program execution speed is highly dependent on vectorization. And vectorization depends on program design, program logic, data structures, coding methods, algorithms, and the nature of the problem being solved. The purpose of this document is to assist the FORTRAN programmer in understanding vectorization and in applying principles and techniques necessary to achieve its benefits. Although many design and programming concepts presented here apply in general to vector computers, much information pertains specifically to vectorization on the two-pipe Advanced Scientific Computer at the Naval Research Laboratory.

Note: Manuscript submitted September 5, 1979.

1. Vectorization

Vectorization is most easily introduced and illuminated by example. Consider arrays A and B, each consisting of 100 numbers. Assume that one wishes to compute array C where $c_i = a_i + b_i$,i= 1,100. The traditional "scalar" computer executes five assembly language instructions one hundred times. There are two memory fetches (a_i and b_i), one addition, one store to memory (for c_i), and an instruction that increments a counter, tests and branches back to load the next pair of input operands. Thus 500 scalar instructions are executed to add arrays A and B. The ASC FORTRAN compiler can generate such scalar object code, if the programmer requests scalar code. The ASC FORTRAN compiler can, instead, generate "vector" object code which executes very differently. Vector code for adding the 100 pairs of operands consists of a single assembly language instruction and an associated table, built by the FORTRAN compiler. The table contains the starting address in central memory of the input and output arrays and the increments for stepping through the arrays, in this case, one. A vector instruction executes by continuously streaming operands from central memory into the central processor, where the addition takes place, and continuously streaming answers back to central memory. The vectorized addition, in this example, may be thought of as 100 additions simultaneously occurring on the hundred pairs of input operands. Actually, during execution, some elements of A and B are being read from central memory, some elements of A and B are undergoing addition in the CP, some answers (array C) are in output buffers, and some are being written to memory.

The above example describes the singly subscripted FORTRAN DO-LOOP:

```
      DO 5 I=1,100
      C(I) = A(I) + B(I)
5     CONTINUE
```

Doubly or triply subscripted arrays in loops nested 2 or 3 levels deep also may be collapsed into a single vector instruction. The loop

```
      DO 5 K=1,10
      DO 5 J=1,10
      DO 5 I=1,50
      C(I,J,K) = A(I,J,K) + B(I,J,K)
5     CONTINUE
```

executes as a single vector instruction which adds 5000 pairs of numbers.

Table 1 lists the time that it takes to execute a DO-LOOP which performs various numbers of additions on the ASC in scalar mode and in vector mode. Vectorized times are given for utilization of one or both arithmetic units (pipes). Times are given in CP clock cycles. A cycle is 80 nanoseconds.

Execution Time for a DO-LOOP for ASC Addition

Length of DO-LOOP	Scalar Time	Vector Time	
		1 pipe	2 pipes
1	60	91	92
10	257	104	113
50	1171	149	139
100	2288	203	170
500	11244	615	402
1000	22437	1154	646

Table 1

Notice that, for scalar execution, the time per addition remains nearly constant whether 10 or 1000 additions are performed. In two-pipe vector mode, the time per addition decreases from 11 units to .65 units as array size increases from 10 to 1000 elements. This timing pattern is characteristic of all vector operations on the ASC. When arrays are large, the benefit from vectorization and from using two arithmetic units becomes very substantial. In this context, increasing array sizes in a given problem, or refining grid resolution is much less costly in computing time for a vectorized code running on a vector computer than is increased array size or refined resolution on a scalar program or a program run on a scalar computer.

The ordinary FORTRAN DO-LOOP representing one operation performed unconditionally on elements of one or two input arrays and producing elements of one output array is the canonical candidate for a vector instruction. Vectorization consists of designing, organizing, and writing programs so that the maximum possible number of arithmetic and logical operations are executed as vector instructions.

2. Programming for Vectorization

Vectorization occurs when a programmer plans to operate on arrays of data instead of individual points of data. Such planning takes place at the program design level, at the subroutine level and at the line level within each subroutine. This section begins at the line-level and progresses to the subroutine and program design level. The programmer typically works in the reverse direction.

2.1 Conditionality

A DO-LOOP is the FORTRAN programmer's idiom for representing operations on an arrays of data. The DO-LOOP

```
      DO 100 I=1,50

      D(I)=A(I)*B(I)+C(I)

100   CONTINUE
```

is compiled on the ASC into two vector instructions; one which multiplies A and B, element-wise, and one which adds the product array to array C, element-wise. The two instructions execute serially; the vector addition follows the vector multiplication. When this loop has an "if test", e.g.

```
      DO 100 I=1,50

      IF(I.EQ.ITEST(I)) GO TO 100

      D(I)=A(I)*B(I)+C(I)

100   CONTINUE
```

it is rejected for vectorization by the compiler. In this example the multiplication and addition may take place on some, but not all, of the array elements. An arithmetic or logical operation is vectorizable only if it is performed unconditionally on all elements of one or two input arrays to produce one resultant array.

Conditionality (the if-test) is often intrinsic to a computation. The following advice, guidelines, and examples suggest ways of achieving significant vectorization in the face of conditionality.

A conditional operation can sometimes be transformed into one which is not conditional. Consider the loop

```
      DO 100 I=1,500

      IF(D(I).GT.XMAX) D(I)=XMAX

100   CONTINUE
```

which tests each element of array D and replaces only those values which pass the test. The equivalent replacement loop

```
      DO 100 I=1,500
      D(I) = MIN (D(I),XMAX)
  100 CONTINUE
```

unconditionally performs the same operation on each element of array D and is thus potentially vectorizable. In fact, the compiler generates one vector instruction which calculates each element of resultant vector D as the minimum value of input element d_i and scalar XMAX. Eliminating the "if-test" and achieving an array operation instead of scalar operations reduces the run time of the above example loop from 857 micro-seconds to 394 micro-seconds.

Now consider code which evaluates one polynomial in case $x \geq 1$, and a different polynomial for $x < 1$.

```
      DO 100 J=1,80
      DO 100 I=1,80
      IF (X(I,J).GE.1.0) Y(I,J)=
     * 1.0+X(I,J)*(2.0+X(I,J)*(3.0+4.0*X(I,J)))
      IF (X(I,J).LT.1.0) Y(I,J)=
     * 1.0+X(I,J)*(5.0+X(I,J)*(6.0+7.0*X(I,J)))
  100 CONTINUE
```

These if-tests may be eliminated by evaluating both polynomials for all values of X and then retrieving the desired resultant value for each x. This is accomplished without conditionality as follows:

```
      EQUIVALENCE (S,IS)
      DO 30 J=1,80
      DO 30 I=1,80
      S(I,J)=X(I,J)-1.0
      U(I,J)=1.0+X(I,J)*(2.0+X(I,J)*(3.0+4.0*X(I,J)))
      V(I,J)=1.0+X(I,J)*(5.0+X(I,J)*(6.0+7.0*X(I,J)))
   30 S(I,J)=LSHF(IS(I,J),-31)
      DO 40 J=1,80
      DO 40 I=1,80
   40 Y(I,J)=S(I,J)*V(I,J)+(1.0-S(I,J))*U(I,J)
```

Scratch array S has values 0 or 1 according to whether $x \geq 1$ or $x < 1$. Function LSHF (logical shift), in this case, replaces each value in S with its sign bit value. Vectorization appears costly in this example since the number of computations is more than doubled. Also, memory is increased since scratch arrays S, U, V are required. Table 2 shows relative execution time for this example before recoding and after. The example has been compiled with both the vectorizing compiler, NX, and the non-vectorizing, non-optimizing compiler, FX, for two sample arrays.

	FX COMPILER	
	20 x 10 Array	80 x 80 Array
Before recoding i.e. with if-tests	1.0	1.0
After recoding i.e. with redundant calculations	2.174	2.325
	NX COMPILER	
Before recoding i.e. with if-tests	.4854	.4673
After recoding i.e. with redundant calculations	.0697	.0482

Table 2

When the NX compiler is used, the recoded version with redundant calculations runs 7 times faster for the smaller array and 10 times faster for the larger array than did the original code. Without vectorization, in the case of the FX compiled versions, this recoding more than doubles execution time since the number of calculations is more than doubled.

An "if statement" within a loop often inhibits vectorization of subsequent statements in the loop which are vector in character. Removing the "if-test" from the loop , or breaking the loop into several shorter loops may result in significant vectorization. When the loop

```
      DO 100 I=1,500
      IF(A(I).GT. AMAXA) A(I)=AMAXA
      C(I)=A(I)*B(I)+A(I)
100   CONTINUE
```

is replaced by two loops

```
      DO 400 I=1,500
400   IF(A(I).GT.AMAXA) A(I)=AMAXA
      DO 500 I=1,500
500   C(I)=A(I)*B(I)+A(I)
```

its time decreases from 1740 micro-seconds to 952 micro-seconds. When conditionality is eliminated totally by using vector library function MIN, the execution time drops to 150 micro-seconds.

Minimizing the ill effects of conditionality is challenging and offers opportunities for creative design and programming. Gains in vectorization are often offset by costs in memory usage and code readability. The degree of vectorization appropriate to a program is decided, individually, for the program, according to its circumstances.

2.2 Subroutine Organization for Array Processing

The fundamental principle for subroutine design is: Plan, organize, and create subroutines which operate on arrays of data instead of points of data. For example, the program

```
      PROGRAM MAIN
      DIMENSION A(100),B(100),C(100)
      DO 20 I=1,100
      CALL SUB1 (A(I),B(I),C(I))
      CALL SUB2 (A(I),B(I),C(I))
      CALL SUB3 (A(I),B(I),C(I))
20    CONTINUE
      END
```

unhappily locks the computation into scalar operations on points a_i, b_i, c_i and requires that the three subroutines be called 100 times each. The DO-LOOP above should be replaced by

```
      CALL SUB1 (A,B,C)

      CALL SUB2 (A,B,C)

      CALL SUB3 (A,B,C)
```

where each subroutine operates on arrays A,B,C. This structure permits vectorized computation and minimizes costly subroutine linkage.

2.3 Vector Math Library Functions

The ASC vectorizing compiler is built to apply the fundamental vectorization principle for subroutines. If a programmer codes

```
      DO 100 I=1,500

      B(I)=SIN(A(I))

  100 CONTINUE
```

the vectorizing compiler collapses the loop into a single call to a vector sine function with input vector A and resultant vector B.

Scalar computers typically have one system FORTRAN library. When a trigonometric function, square-root, maximum/minimum function, etc. is invoked, a point-wise (scalar) function is called with a scalar answer. Vector computers have such scalar functions, and, in addition, typically have a library of vector functions which operate on arrays of points. The Vector Math Pack functions are themselves vectorized. For 500 sine calculations, the time per sine is 16.3 micro-seconds when done in scalar mode and 1.84 micro-seconds in vector mode.

2.4 Algorithms and Mathematical Methods

Vectorization principles governing the choice of algorithms and mathematical methods may be deduced from the line level and subroutine level principles previously discussed. Methods chosen should involve significant unconditional computation on large arrays of data. Algorithms which entail many more arithmetic operations may be preferred over those involving fewer arithmetic opertions which do not vectorize. Inventing new mathematical methods, or rediscovering old ones which were considered computationally inefficient on scalar computers, is currently a rich area of research for numerical analysts and computational physicists.

Recursive computations are intrinsically unvectorizable in their usual form. Consider the recursive relation

```
      DO 100 I=2,100
      A(I)=A(I-1)+B(I)
100   CONTINUE
```

where each element a_i of array A is computed from the element just previously computed, a_{i-1}. A vectorized addition for this loop is equivalent to

```
      DO 100 I=1,100
      AA(I)=A(I)
100   CONTINUE
      DO 110 I=2,100
      A(I)=AA(I-1)+B(I)
110   CONTINUE
```

These loops yield different answers for array A than does the original loop executed in scalar mode. The difficulty is that, in vector mode, newly computed a_i values do not get into the input buffer to be used in computing a_{i+1}. Instead, "old" values of a_i are used to compute a_{i+1}. The ASC compiler flags the original loop as a "vector hazard" and does not generate a vector instruction for it.

When a recursive computation is required, it should be done in a loop by itself, isolated from other calculations. This prevents the vector hazard which it presents from inhibiting vectorization of neighboring calculations.

Techniques exist for coding recursive relations so that they can be safely computed as vectors. Reference 1 is an example of such a technique.

2.5 Memory Management

Vector instructions are most efficiently executed when the elements of operand arrays are stored, in central memory, contiguously with respect to the computation. The FORTRAN code

```
      DIMENSION A(10,50)
      DO 100 I=1,50
      A(K,I)=A(K,I)+B(K)
100   CONTINUE
```

exhibits non-contiguity for input operand A. The FORTRAN dimension statement declares that A is a 2-dimensional array and is stored column-wise in central memory. The addition occurs, element-wise, on a row of A. Thus every 10th value of A as it resides in memory is input and output to this computation.

The ASC hardware memory fetch and store consists of 8 contiguous words, an "octet". If all eight words are data for a vector instruction, the arithmetic unit computes at full speed. If only one or two values in each octet are input for the vector instruction, its execution speed is severely degraded.

The example may be recoded as

```
      DIMENSION A((10,50))

      DO 100 I=1,50

      A(K,I)=A(K,I)+B(I)

100   CONTINUE
```

Double parentheses in the dimension statement (an ASC FORTRAN extension) alert the compiler that A is to be stored row-wise in central memory so that row-wise operations may be done efficiently. Now the loop represents a vector addition on a contiguously stored vector.

2.6 Vector Temporary Space

The vectorizing compiler allocates and manages a variable amount of work space or scratch space in central memory for its own use in generating vector object code. Vector temporary space is re-used by successive vector operations within a subroutine. The temporary space required for a program is the maximum amount used by any one subroutine. The following examples illustrate the compiler's use of vector temporary space. The loop

```
      DO 100 I=1,100

      D(I)=A(I)*B(I)+C(I)

100   CONTINUE
```

represents a vector multiplication followed by a vector addition. The intermediate product array is stored in vector temporary space.

The loop

```
      DO 100 I=1,50
      A(K,I)=A(K,I)*B(I)
100   CONTINUE
```

represents a row operation on a column-stored matrix. The compiler transposes array A in vector temporary space and uses the transposed copy as input and output for the vector multiplication. The vectorizing compiler commonly reverses the order of do-loops or transposes matrices in order to assure contiguity of operands for vector instructions. This latter action may require a significant amount of vector temporary space.

The compiler can allocate vector temporary space only if it knows the lengths of the arrays in question. For example, this loop

```
      SUBROUTINE SMALL (A,B,C,D,N)
      DIMENSION A(N),B(N),C(N),D(N)
      DO 100 I=1,N
100   D(I)=A(I)*B(I)+C(I)
      RETURN
      END
```

is compiled and executed in scalar mode because the compiler does not know array sizes or loop length.

ASC FORTRAN permits a maximal dimension to be declared for variably-sized arrays. Slashes enclose the upper bound. Thus,

```
      SUBROUTINE SMALL (A,B,C,D,N)
      DIMENSION A(N/100/),B(N/100/),C(N/100/)
      DIMENSION D(N/100/)
      DO 100 I=1,N
100   D(I)=A(I)*B(I)+C(I)
      RETURN
      END
```

permits variable dimensions, but gives the compiler an upper bound for the amount of vector temporary space required to vectorize the computation.

The ASC FORTRAN PARAMETER statement can be used to declare PARAMETERS that dimension arrays and serve as loop limits. This technique permits flexibility in changing array dimensions and it declares array dimensions at compilation time so that vector temporary space allocation is straightforward.

Long FORTRAN lines (spanning several cards) require large amounts of vector temporary space for holding intermediate results. This is often wasteful of central memory and sometimes troublesome to the compiler. Although it is not recommended that one operation be coded per line, it is good practice to keep FORTRAN lines reasonably short.

The Vector Math Library functions generally require vector temporary space four or five times the length of the input vectors. If a FORTRAN line contains nested Vector Library functions, management of vector temporary space is a complicated, error-prone task for the compiler.

2.7 Using Two-Pipes (Parallel Processing)

An ASC's central processor may consist of one, two, three, or four pipelined arithmetic units. The NRL computer has two. During program execution, independent instructions are simultaneously executing in the two "pipes". Efficient use of the computer entails optimal usage of both pipes. The optimizing compiler analyzes neighboring arithmetic lines of code and orchestrates pipe utilization for vector operations. If a vector instruction is not immediately preceeded or followed by an independent vector, or a suitable sequence of scalar code, the compiler will generate two vector instructions for the one vector operation so that each pipe can do half the calculations. For example, the vector addition

```
      DO 100 I=1,100
      C(I)=A(I)+B(I)
100   CONTINUE
```

may be partitioned into two vector operations, each doing fifty additions. Vector start-up time is significant enough that it is worthwhile, when possible, to code independent pairs of vector instructions together, thus minimizing partitioning. The compiler's optimization summary reports which vectors are partitioned, so that the programmer can detect and eliminate excessive vector partitioning.

2.8 The Vector Dot Product Instruction

The ASC has a hardware vector instruction which multiplies two one-dimensional vectors and accumulates a sum just like a mathematical inner product (dot product). The following FORTRAN loop pattern is the canonical form recognized by the compiler and collapsed into a single vector dot product instruction.

```
      DO 50 J=1,L
      DO 50 I=1,L
      C(I,J)=0.0
      DO 50 K=1,L
      C(I,J)=C(I,J)+A(K,I)*B(K,J)
50    CONTINUE
```

The ASC vector dot product's significance rests in its speed. Once started, many vector instructions in a CP pipe produce one result per CP clock cycle, (80 nanoseconds). Thus ordinary vector multiplications or additions execute at the rate of 24 million floating point operations per second (MFLOPS). Each output value produced while a vector dot product is executing represents a multiplication and an addition. Thus vector dot product execution speed is 48 million floating point operations per second.

ASC hardware requires that the innermost loop variable for a dot product be a subscript corresponding to contiguous storage for the input vector operand(s). Thus, when A and B are stored column-by-column in central memory, the inner loop

```
      DO 50  J=1,L
      DO 50  I=1,L
      C(I,J)=0.0
      DO 50  K=1,L
50    C(I,J)=C(I,J)+A(I,K)*B(K,J)
```

is incorrectly coded, with respect to matrix A, for a dot product instruction. In this case, the compiler compensates by transposing A in vector temporary space before generating a dot product instruction to perform the matrix-matrix multiply. The programmer is encouraged to store A row-by-row which may be done in ASC FORTRAN, or alternately, to organize data so that line 50 above is written:

```
50    C(I,J)=C(I,J)+A(K,I)*B(K,J)
```

Simple sums of elements of one, two, or three dimensional arrays are recognized by the computer as dot products. Thus loop

```
      DIMENSION A(20,20)
      SUM=0.0
      DO 100 J=1,20
      DO 100 I=1,20
      SUM=SUM+A(I,J)
100   CONTINUE
```

and even the loop

```
      DIMENSION A(20,20)
      SUM=0.0
      DO 100 J=1,20
      DO 100 I=1,15
      SUM=SUM+A(I,J)
100   CONTINUE
```

is collapsed into a dot product instruction.

It behooves an ASC programmer to learn which sums and products can be done as vector dot products and to code them precisely in the form and pattern which the compiler recognizes as a dot product. The vector parameter file, a table built by the compiler to describe and drive the vector instruction, can be examined in case the programmer has confusion or doubt about a specific vector instruction.

2.9 Summary of Programming Principles for Vectorization on the ASC

This list summarizes the principles and guidelines presented earlier in this chapter.

- Plan programs and subroutines which operate on arrays of data instead of points of data.
- Choose algorithms and mathematical methods which are array-oriented and vectorizable.
- Minimize and/or eliminate conditionality.
- Use Vector Math Library functions.

- Do not follow non-vectorizable calculations by vectorizable calculations in the same DO-LOOP.
- Store vector operands contiguously in Central Memory.
- Fix array dimensions at compile time, or, at least give a maximal value.
- Write short FORTRAN lines.
- Code independent pairs of vector operations in neighboring lines.
- Code vector dot products exactly.
- Write nested DO-LOOPS so that the innermost loop moves contiguously with arrays as they reside in central memory.
- Study vectorization summaries and messages from the compiler.

3. Meet the Vectorizing Compiler

The goal of the vectorizing compiler, known as NX, is to transform FORTRAN source code into the sequence of ASC machine language instructions which executes most rapidly in a specific, target ASC central processor. In general, the NX compiler generates more efficient object code than would be coded by an assembly language programmer. Understanding and appreciating the NX compiler requires some knowledge of ASC central processor architecture.

3.1 Brief Description of ASC CP Architecture

Three features distinguish the NRL ASC from its contemporary scalar counterparts. It is a pipeline computer; it has a full set of hardware vector instructions in addition to a full set of scalar instructions; and it is a multi-pipe computer.

An ASC arithmetic unit (AU) is logically and physically organized as a twelve-level pipe. Four levels are devoted to instruction decoding and processing, and eight to arithmetic or logical sub-operations. Thus, when the AU is operating in scalar mode, up to twelve operations are concurrently at some stage of execution. At each CP clock cycle (80 nanoseconds) each arithmetic or logical operation in progress in the pipe drops to a lower level, and one answer may exit to the memory buffer. Pipe levels unnecessary to a particular instruction are bypassed. Operands for calculations and answers are fetched and stored while the calculations are progressing through the pipe.

The most powerful computational capability of the ASC is its ability to run in vector mode. In this situation, a single operation is performed on many pairs of operands. Input values stream continuously into the pipe, an operation is performed in discrete steps within the pipe and answers flow back to central memory at the rate of one per clock cycle. The power of the vector instruction is that it guarantees optimum flow of calculations and data through the pipe.

An ASC may have one, two, three, or four pipes. The NRL computer has two. A user's program is partitioned into independent sequences of instructions which execute independently in the two pipes. Typically the pipes are executing different sequences of code, each with its own buffers of input and output. However, a vector operation may be split so that half the operands are processed in one pipe and half in the other.

3.2 Brief Description of the NX Compiler

The NX Compiler is a large program; it executes slowly on the ASC, and is, therefore, expensive to run. It occupies approximately 32 pages of central memory; a page is 4096 words; it consists of eight overlay structures. The NX Compiler processes approximately 1500 FORTRAN source lines per minute. A second FORTRAN compiler, known as FX, performs a fast, inexpensive compilation and does not optimize code for maximum pipe usage, nor does it generate vector instructions. The FX compiler occupies 28 pages of central memory, is not overlaid at all, and processes approximately 20,000 FORTRAN source lines per minute. The FX compiler is used to detect syntax errors, to debug a program, or in cases when it is desirable to minimize program compilation time at the (often considerable) expense of program execution time.

The NX compiler performs standard machine-independent optimization. In addition, it examines each do-loop and each nest of do-loops to determine whether arithmetic and logical operations within the loop(s) can be done as vector operations. When a FORTRAN construction is inherently vector in character, and is not compiled into vector instructions, the compiler writes a message or suggestion to the programmer. Often the FORTRAN lines may be changed slightly, according to the compiler's suggestion, so that they are recognized as bona fide vectorizable lines, and compiled as such.

The NX compiler substantially re-orders a program's original source scalar code to minimize pipe delays. Differing machine language instructions use and bypass different pipe levels; thus certain instructions may follow other instructions

without delay, while some instruction sequences produce idle "bubbles" in the pipe. The NX compiler orders scalar instructions favorably for pipe usage.

The compiler issues advisories to the Instruction Processing Unit of the Central Processor regarding which vector instructions may execute simultaneously in the two pipes. IPU advisories for each vector also indicate whether or not the vector may execute at the same time as neighboring scalar code executes in another pipe. This involves considerable, sophisticated, independence analysis. The compiler often partitions long vectors so that both pipes may be kept as full as possible.

The NX compiler permits great flexibility in the choice of optimization features. The programmer may turn the compiler's vector generation capability on or off at the program or subroutine level, or at the loop level within a program. Here is a partial list of options available to select or inhibit optimization features of the NX compiler.

I option — An I-level compilation has all vectorization capabilities turned off and nearly all scalar optimization features turned off. It is a basic line-by-line compilation with some scalar optimization.

J option — All vectorization capabilities of the compiler are turned off; however complete scalar optimization takes place, e.g., scalar assembly language instructions are grouped to minimize pipe delays. Local variables are stored in registers to minimize the number of accesses to central memory during program execution.

K option — All scalar optimization of J-level takes place as well as significant vectorization. This is the recommended compilation level for generating object code suited to the ASC CP hardware. Vectors which may be hazardous or which may yield answers different from scalar code are not generated. Lines containing such potential vectors are flagged by the compiler and labeled "hazardous."

L option — Vector hazards in array assignment statements are vectorized, in addition to all optimization of the K and J level compilations taking place. An example array assignment statement is "A=B"

where A and B are arrays. ASC FORTRAN permits such statements. Here is an example vector hazard in an array assignment statement.

```
DIMENSION B(55),A(50)

EQUIVALENCE A(1),B(5)

A=B
```

If this example were done in scalar mode, the assignment of values conforms to scalar loop

```
      DO 100 I=1,50

100   A(I)=B(I)
```

and in particular A(6) would get value B(2), since B(6) occupies the same location as A(2), which was assigned value B(2) at I=2. If the loop were vectorized then A(6) would be assigned the value which was in location B(6) (or A(2)) before the loop was entered. In general this value is different from the value originally in B(2).

When a program contains no array assignment statement hazards, an L level compilation is equivalent to K level.

W option — Vectors are generated for all vectorizable statements regardless of vector hazards. Section 2.4 contains an example of a common vector hazard.

G option — The G option suppresses code re-ordering and is meaningful when used with options J, K, L, or W.

Y option — The Y option forces the compiler to store all local variable values in central memory, instead of holding some of them in registers without memory allocation. Section 3.3.2 contains an explanation of this optimization feature as well as a common pitfall in its use. The Y option is meaningful when used with options J, K, L, or W.

Normally the K option is selected by the ASC programmer. This level permits maximal scalar optimization and allows the compiler to generate all vectors which are safe and reasonable. Care should be taken in selecting optimization options other than K.

3.3 Pitfalls for the Unaware Programmer

Here is a partial list of common programming problems related to usage of the NX compiler.

3.3.1 The Timing Routine Pitfall

A source line consisting of a call to a timing routine appears to the compiler to be independent of neighboring source lines. Thus, when the compiler orders object code for pipe efficiency, a call to a timing routine may be placed in a different place than the programmer intended. If program fragments are timed by such "CALL" statements, an object listing must be perused to determine whether the timing routine is timing the intended, appropriate program fragment.

Re-ordering of a specific source line may be inhibited by placing a statement label on the source line and referencing the label elsewhere in the program. For example,

```
      ASSIGN 100 TO JUNQUE
        .
        .
        .
        .
100   CALL SECOND (1,DTIME)
```

inhibits faulty placement of this call to SECOND.

3.3.2 The Local-Variable-In-Register Pitfall

Some variables which are local only to a specific (sub)routine and which are heavily used in the (sub)routine are held in registers throughout the (sub)routine's execution. A memory cell is not allocated by the NX compiler for such variables. When the (sub)routine is exited, values for such variables are not retained. At subsequent re-entry into the (sub)routine the programmer cannot count on these variables to have the same values they had when the (sub)routine was last exited. A variable in COMMON or in an argument list is not (sub)routine-local. Thus, a recommended way to force memory allocation for a specific variable and avoid this pitfall, is to put the variable in COMMON. Compiling with the Y-option (see Section 3.2) forces memory allocation for all variables, but suppresses a useful optimization feature of the NX compiler.

3.3.3 The 65,535 and 32,767 Limitations

The table which describes and drives a vector instruction, (vector parameter file), has a half-word (16 bit) field to hold loop lengths. Thus, do-loop control variables for vectorized do-loops may not exceed 65,535. For example, the loop

```
      DO 100 I=1,IBIG
100   A(I)=B(I)+C(I)
```

will have a faulty vector parameter file when IBIG is greater than 65,535 at execution time.

Another 16 bit field in the vector parameter table holds an address increment which may be positive or negative. Thus, 32,767 is the maximum size for vectorized arrays in some circumstances. The matrix multiplication

```
      DO 100 J=1,IMAX
      DO 100 I=1,IMAX
      C(I,J)=0.0
      DO 100 K=1,IMAX
100   C(I,J)=C(I,J)+A(K,I)*B(K,J)
```

is collapsed into a legitimate vector dot product when IMAX is less than or equal to 181, but is not when IMAX is greater than 181. (The square of 182 exceeds 32,767).

Reference 6 contains a complete description of the fields in the vector parameter file and their size limitations.

3.3.4 The 511 Vector Parameter File Limitation

A program or subroutine may have at most 511 vector instructions.

3.3.5 Compiler Dumps

The NX compiler is subject to shutdown due to certain user FORTRAN errors and FORTRAN source constructions. A compiler shutdown and dump should be reported to the NRL ASC programming consultant (202-767-3542) who will locate the compiler's difficulty and assist the user in coding around it. It is recommended that NX users compile with the "B" option which halts compilation after a scan phase when-

ever source code contains fatal errors. This prevents the compiler from shutting down during a later optimization or code-generation phase.

3.3.6 Compiler Bugs

A comprehensive ASC bug list is kept by the NRL Research Computation Center. ASC programmers may read this from time to time to note which compiler bugs are outstanding and which have been fixed.

3.4 Conclusion

The vectorizing compiler is a complex and smart ally of the programmer who is writing or converting FORTRAN code to run efficiently on the ASC. The NX compiler vectorizes a large class of FORTRAN source constructions. The programmer must become acquainted with ASC CP architecture, as presented in this Primer, and must understand the goals and concept of vectorization. She or he also must pay close attention to the details of vectorization as reflected in the messages, summaries, and tables output by the compiler.

BIBLIOGRAPHY and REFERENCES

1. J. Boris, Vectorization of Recursive Relations, NRL Memorandum Report 3144, October 1975.

2. G. Cobb, Vectorization from ASC's FORTRAN Compiler, Preliminary Issue, Texas Instruments Incorporated, no date.

3. M. Miller, Efficient FORTRAN Coding for the ASC, Video Tape and Notes, Texas Instruments Incorporated, no date.

4. E. Miner and B. Brooks, Comparative Computer Times Between Vectorized and Scalar Versions of a Large Hypersonic Viscous Shock-Layer Code, Proceedings of the AIAA 11th Fluid and Plasma Dynamics Conference, July 1978.

5. Texas Instruments Incorporated, ASC FORTRAN Reference Manual, Manual #930044-2, January 1976.

6. Texas Instruments Incorporated, Efficient Coding for the ASC Central Processor, Manual #930036-21, January 1976.

7. Texas Instruments Incorporated, FORTRAN Users Guide, Manual #930045-1, July 1977.

ACKNOWLEDGEMENT

The polynomial example in section 2.1 was provided by George Mueller, Naval Research Laboratory.